Model Checking Interrupt-Dependent Software

Colin Fidge
School of Software Engineering and
Data Communications
Queensland University of Technology
Queensland 4001, Australia
c.fidge@qut.edu.au

Phil Cook
School of Information Technology and
Electrical Engineering
The University of Queensland
Queensland 4072, Australia
philc@itee.uq.edu.au

Abstract

Embedded control programs are hard to analyse because their behaviour depends on how they interact with hardware devices. In particular, embedded code typically uses interrupts to respond to external events in a timely manner. Such asynchronous control constructs make static analysis difficult due to the potentially large number of alternative control-flow paths they allow. We show how model checking can be used to effectively analyse the behaviour of interrupt-dependent programs. This is done by developing an abstraction of the code that captures its essential timing and functional properties, including those related to external interrupts. The model is made efficient by grouping program instructions into basic blocks whose behaviour is atomic with respect to interrupts.

1 Introduction

Embedded control programs present significant development and maintenance problems. For efficiency they are often written in low-level languages, which makes them hard to understand. Also, since they interact directly with their hardware environment, they are difficult to analyse in isolation. Moreover, many such systems are *safety critical* due to the influence they may exert on their physical environment.

A particular problem is that embedded programs typically use interrupts to schedule periodic activities or to respond to external events in a timely manner. Such asynchronous control constructs make understanding and analysing the code very challenging, due to the potentially large number of alternative run-time instruction sequences they allow.

Motivated by the challenge of understanding and maintaining legacy avionics code, we show how model checking can be used to effectively analyse an interrupt-dependent control program. This is done by modelling the effect of instructions as transitions on the state of the processor's hardware and that of significant hardware devices. The passage of time itself is also modelled as a state property. The guards that determine when transitions may occur are constructed in a way that allows state-triggered and interrupt-triggered actions to co-exist, with priority given to interrupt actions. Finally, to make the model acceptably efficient, we group instructions into single-entry point, single-exit point basic blocks, provided that the resulting compound actions can be considered atomic with respect to interrupts.

2 Previous Work

The notion of applying model checking to embedded control systems is by no means new. For instance, the SMV model checker was used previously to analyse an abstraction of a medical monitoring system [6]. As in our approach, explicit timestamp variables were used to define when time-dependent events could occur. Unlike our work, however, the Petri net model used as a starting point was not directly related to embedded program code.

Similarly, the SPIN model checker has been applied successfully to a chemical plant controller [4]. Again, the timed automata-based model was derived from a hardware description rather than program code. In this case the main technical challenge was to develop a discrete approximation of the plant's continuous behaviour. To improve the model's efficiency, care was taken to model only those points in time at which significant events occur. We use a similar idea below to avoid generating too many alternative, but essentially equivalent, interrupt behaviours.

More recent, and much closer to our motivation, was application of the LTSA model checker to avionics software [13]. In this case the focus was on component-based software, using a publish/subscribe interaction paradigm. This differs from our interest in legacy programs written in

Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)
0-7695-2465-6/05 $20.00 © 2005 IEEE

the cyclic-executive multi-tasking style. Notably, the resulting LTSA model treated communication as synchronous, rather than asynchronous.

Closest of all to our research was the development of a model checking tool for software running on the Z86 processor [5]. As in our work, the emphasis was on interrupt-dependent programs written in assembly code. Unlike our work, however, the goal was to use model checking to ensure that the interrupt mechanism itself was being used correctly, where we are more concerned with the overall properties of programs that rely on interrupts. The approach used was to model the behaviours of the Interrupt Mask Register, Program Counter and stack for this particular machine in their entirety. The results were impressive, but this approach is much more detailed than we need. For our purposes we are interested in modelling just enough of the interrupt mechanism to be able to make useful predictions about the system's dynamic behaviour. As demonstrated below, we have found that we can achieve practical results with a comparatively abstract representation of the relationship between the Program Counter and the times at which external interrupts occur.

3 Background: Embedded Control Systems

Our research is motivated by the problem of maintaining legacy embedded programs in avionics applications. A typical avionics Mission Computer System consists of several Remote Terminals connected by a central data bus [14]. Each Remote Terminal contains a Central Processing Unit (CPU) and a separate Input-Output Processor (IOP) as shown in Figure 1 [9].

Each Remote Terminal's CPU performs a particular aircraft function (e.g., navigation, displaying cockpit data, deploying weapons, etc), while the IOP provides an interface to other Remote Terminals and peripheral devices in the aircraft. Having a dedicated IOP allows the CPU to execute without having to wait for bus access or for peripheral devices to respond.

The CPU and IOP exchange data via a shared Memory Module. To send data from the Remote Terminal to another terminal or peripheral device, the CPU stores the data in shared memory and instructs the IOP to forward it to the appropriate destination. The CPU can then continue executing while the IOP completes the output operation. To read data from another Remote Terminal or peripheral device, the CPU instructs the IOP to read data from the appropriate source. Then, while the CPU continues executing, the IOP interfaces with the bus or device and places the data received in the shared memory. Once this is done the IOP generates an I/O Completion Interrupt that tells the CPU the input operation has been completed [9].

The software on the CPU typically consists of a set of periodic tasks (processes) controlled by a central Mission Computer Executive which invokes each task at a predetermined frequency [9].

4 Motivational Example

As a simple example of interrupt-dependent code, consider the Altitude Display task in Figure 2. This is intended to be one of the tasks periodically invoked on the Display Processor Remote Terminal [14] when the aircraft is cruising at high altitude. The task's purpose is to send the aircraft's current altitude to the Head-Up Display (HUD) in the cockpit.

At each invocation the task asks for an altitude reading from the aircraft's Radar Altimeter Remote Terminal [14]. While waiting for a response the task estimates the aircraft's altitude by extrapolating from the most recent altitudes dis-

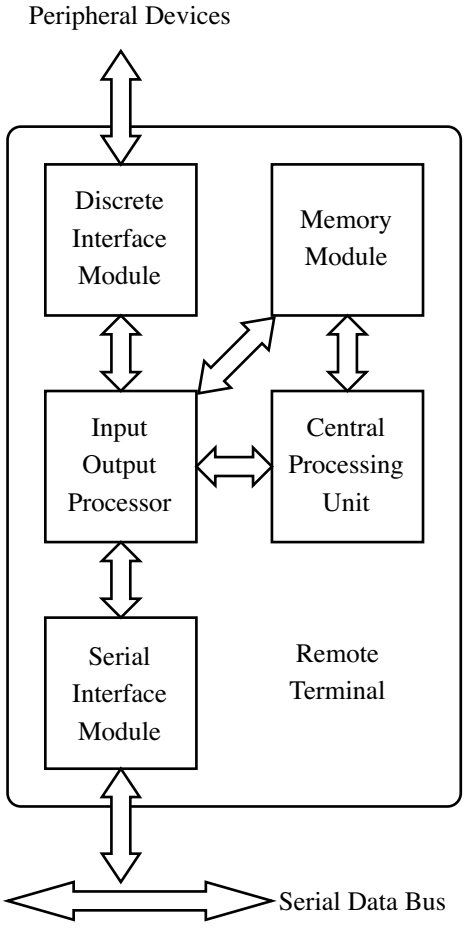

Figure 1. Typical avionics Remote Terminal architecture

```
Label     Operation   Operands          Comment
  isr:    MOV         $TRUE, %R3        Set I/O ready flag
          RETURN                        Resume interrupted task
            ⋮
startio:  MOV         $FALSE, %R3       Reset I/O ready flag
          MOV         $GETALT, %R0      Value that tells the IOP to get an altitude reading
          STORE       %R0, (toiop)      Send instruction to IOP
calcalt:  LOAD        (penalt), %R0     Get penultimate altitude (stored by previous task invocation)
          LOAD        (prevalt), %R1    Get previous altitude (stored by previous task invocation)
          STORE       %R1, (penalt)     Store penultimate altitude for the next invocation of this task
          MOV         %R1, %R2          Copy previous altitude into register 2
          SUB         %R0, %R1          Set register 1 to the difference between the last two altitudes
          ADD         %R1, %R2          Set register 2 to the calculated altitude
display:  BRFALSE     %R3, send         Skip next instruction if no interrupt has occurred yet
          LOAD        (fromiop), %R2    Get altitude reading from IOP
   send:  STORE       %R2, (tohud)      Send altitude to Head-Up Display
          STORE       %R2, (prevalt)    Store previous altitude for the next invocation of this task
```

Figure 2. Assembly program for the Altitude Display task

played. If the altimeter has responded by the time this calculation is complete the altimeter reading is sent to the HUD. Otherwise the estimated altitude is used. The attempt to get the altitude from the altimeter may fail either due to contention for the central data bus or problems with the altimeter itself. Most importantly for our purposes, an I/O Completion Interrupt is used to the tell the Altitude Display task when an altimeter reading has been received.

The program in Figure 2 uses a simple two-address instruction set. The program listing is presented in AT&T-style assembly language: registers are prefixed by '`%`'; immediate values by '`$`'; and direct memory addresses appear in parentheses. For clarity we use symbolic names for constants, including address-valued constants.

The two instructions at label `isr` in Figure 2 form the Altitude Display task's Interrupt Service Routine. It sets register 3 to logical 'true' to indicate that an interrupt has occurred. (More generally, it could also indicate what class of interrupt it was [9].) Not visible in the code is the hardware-implemented action which stores the current Program Counter value and transfers control to label `isr` whenever an interrupt occurs. The `RETURN` instruction subsequently restores the Program Counter and thus transfers control back to the point of the interrupt.

The body of the task is assumed to be invoked periodically by the Mission Computer Executive and begins at label `startio`. Its first actions are to set register 3 to 'false' and then instruct the Input-Output Processor to get an altitude reading from the Radar Altimeter. This is done by putting constant `GETALT` into memory-mapped register `toiop`, which is assumed here to be used by the CPU to send input/output requests to the IOP.

The six instructions beginning at label `calcalt` then perform a dead-reckoning altitude calculation, while waiting for the altimeter to respond. Memory locations `prevalt` and `penalt` contain the previous and penultimate displayed altitudes, respectively, stored by the Altitude Display task's previous invocation. The difference between these two altitudes is added to the previous altitude to produce the estimated altitude for the current invocation.

The four instructions beginning at label `display` then check whether or not the Radar Altimeter has responded yet. If so, the measured altitude is read from shared-memory location `fromiop` and sent to the cockpit Head-Up Display via memory-mapped register `tohud`. If not, the dead-reckoning result is sent to the HUD instead. Control then returns to the central scheduler (not shown).

At first glance the overall strategy embodied by this Altitude Display task seems reasonable. At each invocation it displays either the actual measured altitude or an extrapolation based on the last two altitudes displayed. It would appear that only an unbroken sequence of I/O request overruns could cause the displayed altitude to vary markedly from the actual one. However, as we shall see, our model of this interrupt-dependent task will reveal a serious flaw in the code's dynamic behaviour.

5 Analysing Interrupt-Dependent Code

In essence, model checkers perform an exhaustive state-space search of a state-transition system in an attempt to find a counterexample which refutes a claimed property of the model. To make model checking of an interrupt-dependent program practical, we are therefore obliged to

develop an abstraction of the program which captures all of its important characteristics. However, the overall state space and number of transitions must be kept as small as possible since model checking is notorious for the explosive growth of memory and time it requires for non-trivial examples.

Our approach begins with a pragmatic combination of features from previous formal models of time-sensitive, reactive systems. Significant aspects of the embedded computer's state, such as the contents of registers and significant memory locations, are represented as state variables [10]. A 'current time' variable *Now* is introduced to model the passage of time [1]. A Program Counter variable *PC* is introduced to explicitly control sequencing of instructions [3]. The behaviour of each instruction in the program is represented as a non-deterministic multiple assignment to the system state [11]. Basic blocks of instructions, i.e., groups of consecutive instructions with a single entry and exit point [2], are modelled as single transitions. Auxiliary timestamp variables [1] are added to the model to keep track of the times at which interrupts will occur. The Boolean guards that determine when actions may occur take the state of the Program Counter, the current time and the interrupt timestamps into account [1, 11]. In particular, our handling of guards and the grouping of instructions into blocks is central to our ability to effectively and efficiently model interrupt-dependent code.

For each class of interrupt that may occur our model maintains a timestamp variable that defines when the *next* such interrupt will occur. If the interrupt is currently disabled or otherwise not expected, this time is set to infinity.

Although the sequencing of instructions in an assembly program is primarily governed by the Program Counter, interrupts effectively 'hijack' this flow of control. Therefore, we guard each action performed during the program's 'normal' flow of control with not only the condition that the Program Counter points to the appropriate location, but also that the current time is earlier than all anticipated interrupts, as defined by their timestamps. Conversely, any action that models an interrupt handler is guarded by the condition that the current time equals or is later than the corresponding timestamp. Expressing the guards in this form gives priority to interrupt actions over 'normal' ones.

The most important aspect of our approach is the way we group instructions into basic blocks. Modelling every instruction as a distinct transition, to allow for the possibility that an interrupt occurs during that instruction, would be hopelessly inefficient. Instead we allow basic blocks to be modelled as if they are atomic, even if there is a possibility that an interrupt could occur mid-way through the block, provided that there is no observable difference between the interrupt occuring during the block or immediately afterwards. This is so if the Interrupt Service Routine only assigns to variables that are not accessed by the block. Indeed, good programming style normally dictates that ISRs should be kept short and do little more than set flags, so this is frequently the case.

However, the occurrence of an interrupt during a basic block of instructions will always have an impact on the block's end-to-end execution time. Therefore, we also require that the absolute timing of any 'outputs' from the block cannot have any significant impact on any other part of the system. This is usually the case in multi-tasking control programs anyway. Typically, such programs are calibrated so that only the absolute starting and finishing time of whole task invocations is important, not the timing of actions *within* an invocation [12].

Thus, even though an interrupt may occur *during* a basic block in practice, our model still treats the block as atomic and defers the corresponding Interrupt Service Routine action until after the block has finished, provided the above conditions are satisfied. This is similar to the notion of 'serialisability' of concurrent updates for distributed databases—as long as no observer can see the difference, 'concurrent' actions can be modelled as if they occur in sequence [15].

Of course, this strategy must be applied with care. Nevertheless, informed by an understanding of the program's intended behaviour, and its execution environment, it is usually not difficult to tell which interrupts are enabled during a particular block and whether or not their occurrence can directly influence the block's (functional) behaviour. Moreover, given common idioms of interrupt usage, the analysis and translation described above should be readily automatable.

6 Case Study

To demonstrate this approach we used the Symbolic Analysis Laboratory (SAL), a collection of tools based around an expressive language for describing state-transition systems [8]. In particular, SAL's Bounded Model Checker searches the state space of a model to try to find a counterexample which refutes a given temporal logic formula [7].

The interrupt-dependent program from Figure 2 was modelled in the SAL language as shown in Figure 3. A state-transition system is constructed in SAL from named, guarded actions, written '*name*: *guard* $\rightarrow$ *action*'. The guard is a predicate on the system state and any action with a 'true' guard may be performed. Each action comprises one or more assignments, composed simultaneously. Here we use operator '||' to indicate simultaneous composition. Assignment of an expression E's value to a variable v is written here as $v := E$, and assignment of a value chosen nondeterministically from a set S is denoted $v :\in S$.

Our model of the Altitude Display task consists of six actions. Action *InterruptServiceRoutine* models the effect of the two instructions at label `isr` in Figure 2 (with the interrupt generated by the IOP assumed to consume one instruction cycle). Actions *StartIOSuccess* and *StartIOOverrun* together model the three instructions at label `startio` for the cases where the altimeter will and will not respond in time, respectively. Action *DeadReckoningCalculation* corresponds to the six instructions starting at label `calcalt`. Finally, actions *DisplayMeasuredAltitude* and *DisplayCalculatedAltitude* model the four instructions at label `display` in the situation where the altimeter does and does not respond in time, respectively. (Modelling each of the conditional 'start I/O' and 'display' actions as two separate transitions, with distinct names, makes the traces produced by the model checker easier to read.)

InterruptServiceRoutine:
$Now \geq NextInterrupt \rightarrow$
  $Reg3 := \textbf{true} \;||$
  $MemFromIOP := \cdots \;||$
  $Now := Now + (3 * InstrExTime) \;||$
  $NextInterrupt := \infty$
StartIOSuccess:
$PC = StartIO \wedge Now < NextInterrupt \rightarrow$
  $Reg3 := \textbf{false} \;||$
  $Now := Now + (3 * InstrExTime) \;||$
  $NextInterrupt :\in$
    $\{t : Time \mid Now + (7 * InstrExTime) \leq t \leq Now + (9 * InstrExTime)\} \;||$
  $PC := CalcAlt$
StartIOOverrun:
$PC = StartIO \wedge Now < NextInterrupt \rightarrow$
  $Reg3 := \textbf{false} \;||$
  $Now := Now + (3 * InstrExTime) \;||$
  $NextInterrupt :\in$
    $\{t : Time \mid Now + (9 * InstrExTime) < t \leq Now + (12 * InstrExTime)\} \;||$
  $PC := CalcAlt$
DeadReckoningCalculation:
$PC = CalcAlt \wedge Now < NextInterrupt \rightarrow$
  $Reg0 := MemPenAlt \;||$
  $Reg1 := MemPrevAlt - MemPenAlt \;||$
  $Reg2 := MemPrevAlt + (MemPrevAlt - MemPenAlt) \;||$
  $MemPenAlt := MemPrevAlt \;||$
  $Now := Now + (6 * InstrExTime) \;||$
  $PC := Display$
DisplayMeasuredAltitude:
$PC = Display \wedge Now < NextInterrupt \wedge Reg3 = \textbf{true} \rightarrow$
  $Reg2 := MemFromIOP \;||$
  $MemToHUD := MemFromIOP \;||$
  $MemPrevAlt := MemFromIOP \;||$
  $Now := Now + (4 * InstrExTime) \;||$
  $PC := \cdots$
DisplayCalculatedAltitude:
$PC = Display \wedge Now < NextInterrupt \wedge Reg3 = \textbf{false} \rightarrow$
  $MemToHUD := Reg2 \;||$
  $MemPrevAlt := Reg2 \;||$
  $Now := Now + (3 * InstrExTime) \;||$
  $PC := \cdots$

Figure 3. State-transition model of the Altitude Display task

Other parts of the model, not shown in Figure 3, include: type and constant declarations; variable initialisations; an action which models the way the Mission Computer Executive invokes the Altitude Display task periodically; and the model of the aircraft's dynamic flight characteristics (discussed further in Section 7).

Some features of the program translate directly into the model. These include memory addresses (e.g., address `startio` in Figure 2 becomes constant *StartIO* in Figure 3), general-purpose registers (e.g., register `R2`'s contents are modelled by variable *Reg2*), and values in memory locations (e.g., the contents of memory location `prevalt` are modelled by variable *MemPrevAlt*).

Other features of the model are implicit in the code. Notably, the Program Counter is modelled by variable *PC*, the instruction cycle time is represented as constant *InstrExTime*, and the current absolute time is modelled by variable *Now*.

The model also contains significant features of the hardware environment. Most importantly, the time at which the next I/O Completion Interrupt will occur is modelled by variable *NextInterrupt*, and the way the aircraft's altitude changes between task invocations is modelled by the values assigned to variable *MemFromIOP* (this calculation is not shown in Figure 3). Timestamp *NextInterrupt* is set to infinity by action *InterruptServiceRoutine* to indicate that no further interrupts are expected following this action. Conversely, the interrupt from the IOP is enabled by actions *StartIOSuccess* and *StartIOOverrun*, since these actions model the request sent to the IOP that initiates that chain of events leading to an interrupt.

Curiously, whether or not the interrupt arrives in time or overruns is predetermined in the model. Action *StartIOSuccess* chooses a value for timestamp *NextInterrupt* that guarantees the IOP will respond in time, whereas action *StartIOOverrun*'s occurrence always results in the IOP failing to respond before the dead reckoning calculation is completed. (In Figure 3 the range of times within which the I/O Completion Interrupt may occur is expressed in terms of the

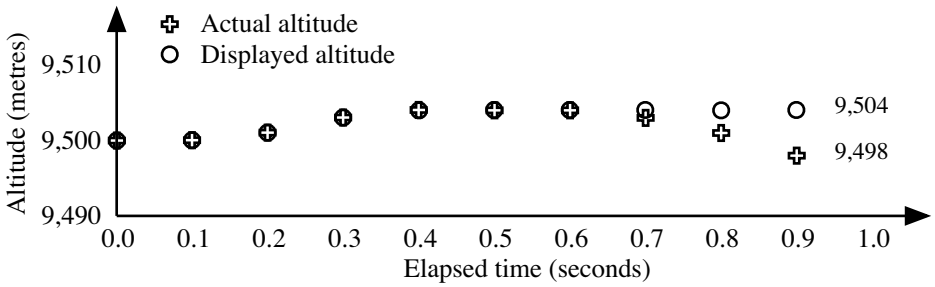

Figure 4. Counterexample produced by the model checker showing an anticipated failure mode

number of instructions executed, starting from the one at label `startio`, inclusive.) Thus the model checker's choice between these two actions determines which of the two actions *DisplayMeasuredAltitude* and *DisplayCalculated-Altitude* will occur subsequently.

The Boolean guards on each action take the relationship between transitions that are enabled by the Program Counter's value and by the occurrence of asynchronous interrupts into account. Each 'normal' action is guarded by the requirement that *PC* equals the corresponding code block's starting address and that the current time *Now* is earlier than any forthcoming interrupt. Conversely, action *InterruptServiceRoutine* will occur whenever *Now* equals or exceeds *NextInterrupt*. This means that action *Interrupt-ServiceRoutine* will always occur as soon as possible.

Most importantly, we need to ensure that our grouping of instruction sequences into atomic actions accurately preserves the program's behaviour with respect to interrupts. Actions *StartIOSuccess* and *StartIOOverrun* can be treated as atomic because they occur before the I/O Completion Interrupt is enabled. Action *DisplayMeasuredAltitude* can be treated as atomic because it is performed only after the interrupt has arrived.

However, if the interrupt arrives in time it may occur while the sequence of six instructions at label `calcalt` in Figure 2 is executing. (The timestamp assigned to variable *NextInterrupt* in action *StartIOSuccess* ensures this in the model.) Nevertheless, these six instructions can be modelled as the atomic action *DeadReckoningCalculation* in Figure 3 because none of the variables *Reg3*, *MemFromIOP* and *NextInterrupt* set by action *InterruptServiceRoutine* are accessed by the assignments in *DeadReckoningCalculation*. Thus it does not matter whether the Interrupt Service Routine is modelled as occurring during or immediately after this block of instructions.

However, this is not true of the current time variable *Now,* which is both set and accessed by all actions. Therefore, we must also be sure that the absolute timing of events within action *DeadReckoningCalculation* is not significant for any other part of the system. The only variables assigned by this action are registers, whose values are meaningful only to this invocation of the Altitude Display task, and the value in memory location `penalt`, which will not be accessed again until the next invocation of this task. Thus, we can safely assume that the absolute times at which these variables are set is not significant. (We assume that no other task reads from location `penalt`. Even if one did, it could not make reliable use of the specific time at which the Altitude Display task writes to the location.)

Similarly, if the I/O Completion Interrupt arrives too late (as defined by action *StartIOOverrun*) it will occur during action *DisplayCalculatedAltitude*. Again, none of the assignments in this action relies on variables set by the Interrupt Service Routine (other than *Now*).

However, action *DisplayCalculatedAltitude* assigns to memory location `tohud` which we assume *is* observable externally, so we must also consider the potential effect of the Interrupt Service Routine executing before versus after this externally observable event. As mentioned above, a real-time task set is usually designed in such a way that 'intra-task' event timing is insignificant compared to the overall schedule of whole task invocations [12]. Therefore, provided that the instruction cycle time is much smaller than the Altitude Display task's period, we can safely treat action *DisplayCalculatedAltitude* as atomic.

7 Practical Results

To confirm the accuracy of our approach we used SAL's model checker and the model in Figure 3 to analyse the dynamic behaviour of the program in Figure 2. Our aim was to test the claim that the displayed altitude always stays within 5 metres of the actual altitude (as measured by the radar altimeter).

The model of the aircraft's dynamic behaviour (not shown in Figure 3) was calibrated on the assumptions that: the aircraft is in 'cruise mode' at an altitude between 9,000 and 10,000 metres; the Altitude Display task is invoked with a frequency of 10Hz; the CPU's instruction cycle time is 5 microseconds; the aircraft's maximum vertical velocity is

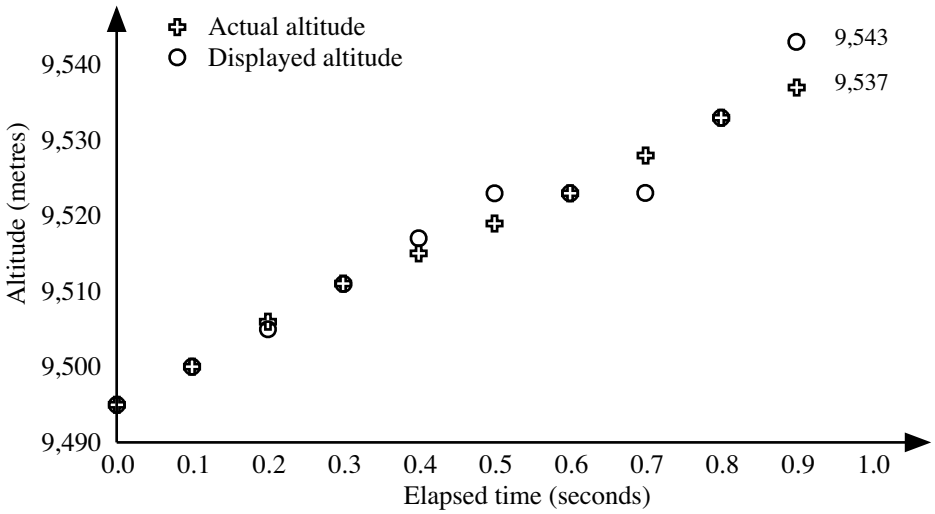

Figure 5. Counterexample produced by the model checker showing an unexpected failure mode

360km/hr; and its maximum vertical acceleration is 10m/s^2.

Given the constraint on the aircraft's vertical acceleration, we guessed that it would require three consecutive I/O request overruns for the displayed altitude to vary by 5 metres from the measured one. Indeed, when we allowed I/O overruns to occur without bound, the model checker produced a counterexample that exactly confirmed our intuitions as shown in Figure 4. In this trace the I/O Completion Interrupt arrives too late in the task cycles ending at times 0.7, 0.8 and 0.9 seconds. Since the aircraft begins descending during this interval, following a period of level flight, the extrapolated altitude diverges from the actual one until they differ by 6 metres in the cycle beginning at time 0.9 seconds.

We then placed a limit on the number of consecutive I/O request overruns allowed by the model, expecting that this would ensure the displayed altitude remains acceptably accurate. To our surprise, however, the model checker produced the counterexample shown in Figure 5, where I/O overruns occur in the cycles ending at times 0.2, 0.4, 05, 0.7 and 0.9 seconds. Even though there are no long sequences of consecutive overruns, and the aircraft's vertical velocity is almost constant, the displayed altitude still goes out of range!

This unanticipated behaviour is due to the displayed altitude's ability to oscillate around the actual (measured) altitude with increasingly large errors. The aircraft accelerates upwards in the cycle ending at simulation time 0.2 seconds which causes the task to underestimate the altitude slightly. The aircraft then accelerates downwards in the cycles ending at times 0.4 and 0.5 seconds, which causes the task to overcompensate in the other direction, and so on. Thus a sequence of (gentle) vertical accelerations, coupled with simultaneous I/O request overruns, allows the absolute difference between the actual and displayed altitudes to grow over time: -1 metre at simulation time 0.2 seconds, $+4$ metres at 0.5 seconds, -5 metres at 0.7 seconds and finally $+6$ metres at time 0.9 seconds.

This example thus not only confirmed the accuracy of our approach to modelling interrupts, but also revealed that the program code in Figure 2—although seemingly correct on first inspection—has an unacceptable dynamic behaviour. (The inherent instability of this type of process is well-known in control theory, of course. We can compensate for it by basing the extrapolation on several preceding altitudes, not just the last two.)

8 Conclusion

Embedded control programs typically use interrupts to achieve real-time responsiveness, but asynchronous control constructs make program analysis difficult. Motivated by the need to understand and maintain legacy control systems, which may be safety critical, we have illustrated a practical technique for model checking interrupt-dependent programs. This was done by carefully modelling the relationship between state-triggered and interrupt-triggered transitions, and by considering the atomicity of instruction sequences with respect to interrupts. As shown by the case study above, the approach is accurate enough to reveal subtle aspects of a control program's dynamic behaviour. In future work we will explore further abstraction techniques needed to make larger examples acceptably efficient, including multi-task examples.

Acknowledgements We wish to thank the anonymous reviewers for their encouraging feedback. This research was funded by Australian Research Council Discovery-Projects grant DP0449773, *Verified Emulation of Legacy Mission Computer Systems*.

References

[1] M. Abadi and L. Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Language and Systems*, 16(5):1543–1571, Sept. 1994.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] R.-J. R. Back. Refinement of parallel and reactive programs. Technical Report Caltech-CS-TR-92-23, California Institute of Technology, 1992.

[4] E. Brinksma and A. Mader. Model checking embedded system designs. In *Proceedings of the Sixth International Workshop on Discrete Event Systems (WODES'02)*, Oct. 2002. Extended Abstract.

[5] D. Brylow, N. Damgaard, and J. Palsberg. Static checking of interrupt-driven software. In *Proceedings of the International Conference on Software Engineering (ICSE'01)*, pages 47–56, 2001.

[6] L. Cortés, P. Eles, and Z. Peng. Formal coverification of embedded systems using model checking. In *Proceedings of the 26th Euromicro Conference*, pages 106–113, Sept. 2000.

[7] L. de Moura. SAL: Tutorial. Technical report, SRI International, Apr. 2004.

[8] L. de Moura, S. Owre, and N. Shankar. The SAL language manual. Technical Report SRI-CSL-01-02 (Rev. 2), SRI International, Aug. 2003.

[9] J. D. G. Falardeau. Schedulability analysis in rate monotonic based systems with application to the CF-188. Master's thesis, Department of Electrical and Computer Engineering, Royal Military College of Canada, May 1994.

[10] C. J. Fidge. Verifying emulation of legacy mission computer systems. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 187–207. Springer-Verlag, 2003.

[11] C. J. Fidge. Formal change impact analyses for emulated control software. *Software Tools for Technology Transfer*, 2005. To appear.

[12] R. Gerber, S. Hong, and M. Saksena. Guaranteeing real-time requirements with resource-based calibration of periodic processes. *IEEE Transactions on Software Engineering*, 21(7):579–592, July 1995.

[13] Z. Gu and K. G. Shin. Model-checking of component-based real-time embedded software based on CORBA event service. In *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC 2005)*, May 2005.

[14] C. D. Locke, D. R. Vogel, L. Lucas, and J. B. Goodenough. Generic avionics software specification. Technical Report CMU/SEI-90-TR-8, Software Engineering Insititute, Carnegie Mellon University, Dec. 1990.

[15] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.